N89-15568

AN OVERVIEW OF VERY HIGH LEVEL SOFTWARE DESIGN METHODS

Maryam Asdjodi and James W. Hooper
Computer Science Department
The University of Alabama in Huntsville
Huntsville, Alabama, 35899

ABSTRACT

Very High Level design methods emphasize automatic transfer of requirements to formal design specifications, and/or may concentrate on automatic transformation of formal design specifications that include some semantic information of the system into machine executable form.

Very high level design methods range from general domain independent methods to approaches implementable for specific applications or domains. Applying AI techniques, abstract programming methods, domain heuristics, software engineering tools, library-based programming and other methods different approaches for higher level software design are being developed. Though one finds that a given approach does not always fall exactly in any specific class, this paper provides a classification for very high level design methods including examples for each class. These methods are analyzed and compared based on their basic approaches, strengths and feasibility for future expansion toward automatic development of software systems.

INTRODUCTION

Automatic programming is one of the long range goals of computer science research. Understanding the natural language interface, converting the specifications in natural language to formal design specifications, and developing implementations are constituent components of automatic programming [2]. Natural language understanding has been an evolutionary process. In its actual implementation, automatic programming always is viewed as substitution of a higher level language for specifying a system to a machine for the languages that are presently available [14]. In order to avoid ambiguity and make the problem manageable, a limited set of vocabulary and interpretation rules are used for the machine interface. Compilers are among the primary tools that improved software specification and introduced basic generic and reusable programming concepts (e.g., loop structures). They allowed higher level specifications than what a machine by its nature was designed to understand. Specification of a software system in a high level language, should be based on specific syntactic rules (BNF) of the language. Compilers are designed to verify software specification (i.e. program) correctness by detecting mainly the

syntactic errors of the implementation and to develop executable specifications (i.e. machine code) for correct programs. Syntactic errors are not the only inaccuracy of programs. Logical and semantic errors result in a larger class of faulty programs. Semantic information includes the definition of objects, relations, rules, and algorithmic concepts that are used for describing the system. Errors related to these interpretations usually are referred to as semantic errors [13]. Semantic errors result from misrepresentation or misunderstanding of the meaning of the requirements or design parameters. Compilers for high level languages such as FORTRAN detect few of these errors.

Very high level (VHL) design methods are being developed by moving up toward greater abstraction of specifications and automatic software generation by relaxing syntactic rules of high level languages, and/or including more semantic information in design specifications. The designer's knowledge about the real system is represented by different methods. Object-oriented programming incorporates a view of real-life entities in terms of their functions and relations with other entities. Logic-based programming models a system in terms of logical statements and assertions. Application of artificial intelligence methods for designing software systems is recommended for use by software engineers [17, 20]. Transformation techniques are used for converting VHL design specifications into implementations. Knowledge-based systems are used for defining an application domain to a computer. What is common in all of these approaches is the necessity of more generic and adaptable constructs for VHL specification of a software system. These reusable aspects of VHL design tools range from standard methodology and control structures of design to generic objects and library components. The next section provides a review of VHL design methods and their approach to reusability.

CLASSIFICATION OF APPLICATION

The design of a software system refers to specification of its algorithmic concepts, data structures, functional components, and interfaces between these components [12]. It is the most important and crucial phase of the software life cycle. Adaptable and more abstract designs, when automatically transformed to implementation in high level languages, release the software system designer from dealing directly with the syntax of programming languages, resulting in more reliable implementations. Different VHL design approaches emphasize reusability of specification, structure, and methodology of the software design, in a different level. They range from efforts to develop generalized structural design methods for transforming informal requirements of problems to formal design specification, to approaches for implementing predefined design elements. Although VHL design approaches are very diverse, they are grouped in the following categories with respect to their major approaches.

- General approaches.
  - Software engineering approaches.
  - Program transformation approach.
  - Component composition approach.
- Application-oriented Methods.
  - Knowledge-based approach.
  - Application language approach.
  - Object-oriented programming.

GENERAL APPROACHES

General VHL design methods provide means for designing a system by applying design languages, environments and tools that are independent of the application domain. General VHL design methods allow more validation of specification of designs by implementing general programming and software design knowledge for developing VHL specifications and transforming them to software. In most cases logical, functional, or relational design approaches are enforced by general VHL design methods. Generally these systems are interactive and no knowledge of any application field is required. The following subsections describe classes of approaches in this category.

Software Engineering Approaches

Software engineering emphasizes systematic development of software systems. Complete development of life cycle phases, including requirements, design, implementation, testing and maintenance, as well as traceability between these phases, is encouraged. Design tools are developed to enforce a uniform structure for specifying the system design, that can be traced up to requirements and down to implementations. Design tools are usually supported by standard methodologies for designing a system, by means of very high level design languages, menus, tables, and graphic notations. Some software engineering tools specify a system in terms of objects, and their relationships and attributes. For each functional component, interface conditions in terms of data and control flow and relationships with other components are given. This information is used for verification and consistency checking and tracing among components of the design. Generally a specific design and control structure is enforced by the tool. For example HOS (Higher Order Software) applies a hierarchical structure [7] and a state-based structure is suggested by Matsumoto [12]. HOS transfers design specifications represented by the functional language AXES to programs in high level languages. In HOS each system is represented by mathematical functions, each function having a specified domain of inputs and range of outputs. A control map is used for interface checking among levels of the functional specifications. Static simulation is used for verification of specifications, and a dynamic simulator provides means for simulating execution of HOS programs. HOS facilitates two levels of transformation, from requirements to design and from design to implementation.

Program Transformation Approach

The program transformation method provides for stepwise refinement and transformation of functional or logical specifications of a system to the implementation. The methods used for the refinement of specifications include rule deduction, theorem proving, and pattern matching. Refinement methods may result in huge amounts of intermediate results. Source-to-source transformation rules are used to simplify and optimize the refinement process. Abstract specifications provide very high level programs at the root of a refinement tree, and applying refinement and source-to-source transformation rules, customized application programs may be provided in a high level language as the leaves of the tree. This method is also sometimes called the stepwise refinement method. Program transformation methods share refinement and transformation methods with different areas of computer science such as artificial intelligence, knowledge-based programming, rapid prototyping, and optimization techniques for compiler construction.

Goldberg [6] has summarized techniques that are applied in program transformation approaches as follows. Stepwise refinement rules mainly include folding and unfolding VHL specifications with the lower level specifications, possibly adding conditions for clarifying VHL concepts in terms of implementations in a high level language. Source-to-source transformations applied for simplification of refinement process including loop optimization, finite differencing, assertion maintenance, algebric or logical simplification, and storage efficiency methods.

The stepwise refinement method is used in the CHI system, [18]. In the CHI system, the language V is used for specification of the design of the system using logical, very high level structure. Logical expressions in the V language, using a pool of generic and instantiated objects, are refined to the lower level constructs of the V language, and finally to LISP. Logic assertion compiler and Rule compiler are used for source-to-source transitions and refinement of specification to the lower level constructs. A data structure synthesizer is used to provide a LISP implementation from generic data objects.

Component Composition Approach

Component composition techniques provide for combination and customization of components from a library of generic components. A system is designed by invoking and interfacing library components and reusing predesigned components. Component composition techniques represent reusable design in its precise and true sense. Due to the fact that a library should be searched for the right component, this method also is referred to as programming by inspection [16]. The adaptable components may be objects representing primitives of the language (e.g., data structure operations, control facilities), and "modules", "plans" or "packages" representing more complex components (i.e.,

frequently-applied generic modules). For each component some information is provided, such as name, description of functionality, parameters, interface conditions, and rules or axioms for application. A vocabulary set is required for communication between the user and the system for recognition of the library components. Selected components are customized and instantiated by evaluation of their axioms, interface conditions, and generic parameters. Usually a system is designed by decomposition in a top-down fashion to the basic functional components. In order to design a software system the component composition method is used in a manner similar to the bottom-up programming method. Low-level components are customized and combined to provide more complex components from which the last one is the software system. In general the major requirements for implementing this approach include generic design of components, a library, and customization and combination methods.

Numerous studies about human factors in algorithm design and computer programming have suggested that the component composition methods are very close to the human approach [1, 19]. An example of the component composition approach is presented by Goguen [5] in the Library Interface Language (LIL). The language uses very high level generic packages, applying equational logic expressions. Generic packages satisfy "Theories" for their input parameters. Theories provide interface conditions and/or properties of the parameters of the other entities. "Views" show how a given entity (i.e. a package) satisfies a Theory. Finally, the instantiation phase binds the formal parameters to the actual programming language (Ada) data structure. A LIL program is developed by combining, modifying, and importing, using packages or some of their parameters.

APPLICATION-ORIENTED METHODS

Application oriented methods apply reusable designs for producing software systems within a specific application or domain. Applying the domain-specific analysis and software design conventions provides for generation of more efficient software for the domain. Design elements developed in some of these approaches are adaptable in the sense that they represent or apply some classes of objects of the domain.

Knowledge-Based Approach

Knowledge-based methods use domain rules and knowledge, in conjunction with general methods for interpreting the input specifications of a system, and provide some formal or executable form of specifications. Domain analysis may be represented in terms of the software components [11], methods of generating them, theories, rules and experimental facts, domain-dependent refinement rules of specifications, technical names and

concepts, and the taxonomy of the domain. This analysis may be used to provide a library of generic components for the domain, for transforming and refining specifications, or for providing methods for deriving more efficient implementations. Though this approach also requires some syntax for input description, requirements are frequently achieved by interactive guidance by the user, using a domain-dependent vocabulary. An important factor about knowledge-based design methods is the role of heuristics in applying domain knowledge and in designing and developing systems. This results in a wide variety of approaches for introducing and applying adaptable designs. An example is an automatic software development system for oil drilling purposes, developed by Schlumberger-Doll Research [3]. The system originally was a problem solver to develop software for solving oil well logging problems. Problem specification is given by a computationally-naive user applying concepts and terms of the domain. Applying stepwise refinement methods and user-defined informal specifications, the system produces a formal design and finally software. Domain knowledge is used for maintaining classification of problems and solutions, recognizing the class of input specification, and providing refinement rules to obtain formal design specifications and implementations.

Application Language Approach

Programming languages use a set of vocabulary and parsing rules to interpret the design of a software system. Tools like lexical analyzers, parsers, and interpreters are based on programming language rules (e.g., BNF), and are used for transforming high level problem representations to machine level code. Software systems developed for specific application domains usually have a set of common concepts including functions, objects, and even problem analysis. These common concepts are used in the syntax of application-oriented languages to allow specifications at a level higher than ordinary programming languages. Similar techniques to the conventional language techniques are used for translation of the programs in application-oriented languages into lower level programs in a programming language. An example of such languages is the simulation language SLAM [15]. SLAM accepts simulation programs and translates them to programs in FORTRAN, and like most other simulation languages has predefined features such as time management, arrival distributions, limited-resource management, and performance data collection. Other examples are graphic languages (packages) that allow higher level descriptions of geometric objects.

Object-Oriented Programming

Different programmers approach software design problems differently. The functional decomposition method emphasizes actions, while data interaction is used as the primary focus for

designing a data-centered system. Considering both approaches simultaneously, object-oriented programming views a system or a domain as a collection of objects and their interactions along with their primary functions (methods). This approach allows programming in problem domain concepts rather than machine-oriented programming in terms of variables, memory addresses, operators and operands. Most software design methods somehow deal with objects, their related functions and attributes [9]. Simulation languages come very close to implementing objects and their functions in the manner of object-oriented programming (actually the simulation language SIMULA is considered to be one of the predecessors of the object-oriented languages). The most common definition of an object is an encapsulated data type which can only be accessed through its defined functions or methods [4]. The internal structure of an object is hidden from its users and its functions provide a shell for it. Usually a "message" is used to communicate with an object and to request execution of any of its functions. Most Algol 60 descendant languages that allow definition of data types have the capability to define objects. Encapsulation, concurrent message execution, generic objects, inheritance of objects and methods, libraries of objects, and graphic user-friendly depiction of objects are among the built-in features in the recent object-oriented languages.

Though we have classified object-oriented programming as an application-oriented approach (due to its highly domain dependent application), conceptually it is a general method for designing software systems for any domain. The SMALLTALK language and environment is an integrated system designed on the basis of the object-oriented approach [10]. Everything in SMALLTALK is an object, from numerical types like integers up to entities of the operating system like windows. It allows concurrent message execution for objects of a class, and uses automatic garbage collection for deallocation of resources that may be dynamically bound by messages and are not referenced any longer.

ASSESSMENT OF VERY HIGH LEVEL DESIGN APPROACHES

Software design methods are evaluated from different perspectives. Efficiency, reliability, complexity, degree of automation, and reusability are among the factors that are used here to assess VHL design technique. Emphasis placed by different VHL design methods on each of the above factors varies greatly. Program transformation, in general, requires the user to be able to apply a logical-based or functional-based language. The refinement and transformation process of logical or functional specification is by nature very inefficient [8]. Rule-based refinements require substantial time and storage, and develop huge intermediate results. Refinement deadlock (an intermediate result for which there is no refinement) is another drawback for the program transformation approach. In order to provide a more user friendly environment for obtaining specifications from the user, interface languages are used and

translated to the logical/functional design language. This results in a less efficient procedure (compared with other methods) for implementation of the system. In spite of implementation inefficiency of logical or functional-based specifications, the program transformation approach automatically develops full verified implementations and is best suited for verification of designs and for rapid prototyping.

The software engineering approach is based on independent generation and verification of life cycle phases. Specifications at the requirements level can be traced to the design and implementation levels. Design tools are used to standardize design and control structure, and provide reusability of design methodology and structure. Most design tools emphasize interface checking and verification of design specification but do not provide implementation.

Systematic software generation through specification of systems in life cycle phases has been considered in other research than the software engineering approach, per se. Program transformation techniques tend to apply life cycle concepts in their methodologies. Interface languages in these systems play the role of requirement languages and provide consistency checking. On the other hand HOS, one of the very few software engineering tools that claim automatic software generation, implements a functional design language and includes some of the characteristics of the program transformation method. Similar to the component composition method, HOS applies a library of modules for generation of software.

Component composition methods provide efficient means for developing implementations. Considering the degree of reusability and application of predesigned features, the component composition method is preferred to the other general design techniques, especially if combined with knowledge of an application domain. Another advantage of the component composition method is that the internal representation of reusable components can be hidden from the user of these fragments. For example a logic-based language may be used for internal implementation of library components, while the user may use some simple syntax similar to natural language for implementation and instantiation of these components. As mentioned above this is not the case for the program transformation approach. Generic components can be compiled into machine language and saved in the library. These stand-alone standard library components are also referred to as "software ICs" [4]. Like hardware ICs, software ICs can be independently tested, documented, and used for different applications. Hardware ICs, as reusable and encapsulated functional units, have resulted in a revolution for hardware productivity. Though reusable library components may not result in the same revolutionary progress, their application is a milestone in the evolution of the software industry.

Application-oriented approaches in general provide more efficient software for the domain. The interface language applied for specification of the system is closer to the natural languages and applies concepts of the domain. Consequently it is more convenient for users who are familiar with the application domain. The degree of automation, efficiency, and degree of reusability of knowledge-based methods depend on the method used (component composition or program transformation) and the heuristics applied for representing the knowledge of the domain. Some of these systems concentrate on reusability of domain components and improving the productivity of the software generation process [11]. Others emphasize automation and provide rule-based deduction for automatic software generation [3].

Domain language-based design methods allow high level specifications in terms of domain concepts and have resulted in much more efficient implementations. The disadvantages of these languages is their closed view of the application domain. The sets of domain concepts and interpretations are fixed, and language interpreters and parsers have a fix understanding of the domain, which is not extendable. Object-oriented methods, like domain specific languages, allow programming in terms of domain concepts, though they are not as efficient as domain languages. Pure object-oriented programming encapsulates objects, consequently any higher level function needs to be a combination of methods of objects. The resulting code usually is not very efficient and needs optimization.

CONCLUSION

In view of the above comparative analysis, we have become convinced that the greatest practical leverage for reuse can come by a combination of the component composition and application oriented approaches. Component composition methods in general are capable of supporting development of new and complex components from the existing library components more efficiently than other general design methods and can grasp the essence of object oriented programming (that is, designing software in terms of domain concepts), and can enhance the approach and improve its efficiency.

The idea of creation of a single very high level design tool that develops efficient programs for every application domain does not seem to be practical. Representation of programming knowledge in general is not sufficient or efficient for all application domains. Combination of knowledge of application domain and component composition approach develops an open environment for higher level and domain related design of software systems and is thus a step closer to automatic programming.

REFERENCES

1. Adelson, B. and E. Soloway. 1985. "The Role of Domain Experience in Software Design." IEEE Trans. on Software Engineering, Vol. SE-11, no. 11 (Nov.): 1351-1360.
2. Barr, A. and E.A. Feigenbaum. 1982. The Handbook of Artificial Intelligence, Vol. 2. William Kaufman Inc.
3. Barstow, D.R. 1985. "Domain-specific Automatic Programming." IEEE Trans. on Software Engineering, Vol. SE-11, no. 11 (Nov.): 1321- 1336.
4. Cox, B.J. 1986. Object Oriented Programming An Evolutionary Approach. Addison Wesley.
5. Goguen, J. and M. Moriconi. 1987. "Formalization in Programming Environment." Computer, Vol. 20, no. 11 (Nov.): 55-64.
6. Goldberg, A.T. 1986. "Knowledge-Based Programming: A Survey of Program Design and Construction Techniques." IEEE Trans. on Software Engineering, Vol. SE-12, no. 7 (Jul.): 752-768.
7. Hamilton, M. and S. Zeldin. 1979. "The Relationship Between Design and Verification." The Journal of Systems and Software, Vol. 1, no. 1, 29-56.
8. Hoare, C.A.R. 1987. "An Overview of Some Formal Methods for Program Design." Computer, Vol. 20, no. 9 (Sep.): 85-91.
9. Hooper, J.W. 1985. "BPL: A Set-Based Language for Distributed System Prototyping." International Journal of Computer and Information Sciences, Vol. 14, no. 2, 83-103.
10. Key, A. and A. Goldberg. 1977. "Personal Dynamics Media." Computer, Vol. 10, no. 4, (Apr.): 31-41.
11. Lanergan, R.G. and C.A. Grasso. 1984. "Software Engineering with Reusable Design and Code." IEEE Trans. on Software Engineering, Vol. SE-10, no. 5 (Sep.): 498-501.
12. Matsumoto, Y. 1984. "Some Experience in Promoting Reusable Software: Presenting in Higher Abstract Levels." IEEE Trans. on Software Engineering, Vol. SE-10, no. 5 (Sep.): 502-512.
13. Pagan, G.F. 1981. Formal Specification of Programming Languages: A Panoramic Primer. Prentice-Hall.
14. Parnas, D.L. 1985. "Software Aspects of Strategic defense Systems." American Scientist, Vol. 73, no. 5 (Sep.): 432-440.
15. Pritsker, A.A.B., and C.D. Pegden. 1979. Introduction to Simulation and SLAM. Halsted Press, a Division of John Wiley & Sons, Inc..
16. Rich, C. 1984. "A Formal Representation for Plans in the Programmer's Apprentice." M.L. Brodie, J. Mylopoulos, and J.W. Schmidt (eds) On Conceptual Modeling, Chapter9. Springer-Verlag.
17. Simon, H.L. 1986. "Whether Software Engineering Need to Be Artificially Intelligent." IEEE Trans. on Software Engineering, Vol. SE-12, no. 7 (Jul.): 726-732.
18. Smith, D.R., G.B. Kotik, and S.J. Westfold. 1985. "Research on Knowledge-Based Software Environments at Kestrel Institute." IEEE Trans. on Software Engineering, Vol. SE-11, no. 11 (Nov.): 1278-1295.
19. Soloway, E. and K. Ehrilich. 1984. "Empirical Studies of Programming Knowledge." IEEE Trans. on Software Engineering, Vol. SE-10, no. 5 (Sep.): 595-609.
20. Tichy, W.R. 1987."What Can Software Engineers Learn from Artificial Intelligence?" Computer, Vol. 20, no. 11(Nov.):43-54.